Chapter 2

Network Flows

Flow problems are among the best-understood problems in combinatorial optimization. They are rather important because of their numerous applications.

2.1 Maximum Flows and Minimum Cuts

A *network* is a (simple) digraph $G = (V, A)$ where each edge has a *capacity* $c : A \to \mathbb{R}^+$ and we have two distinguished vertices, the *source* s and the *sink* t. We often write $N = (G, c, s, t)$.

For any vertex v, let $\delta^-(v)$ be the set of *incoming edges* of v, i.e., $\delta^-(v) = \{uv \in A : u \in V\}$ and $\delta^+(v)$ the set of *outgoing* edges of v, i.e., $\delta^+(v) = \{vu \in A : u \in V\}$. Let $f : A \to \mathbb{R}^+$ be any function on the edges. Define the *balance* $\mathrm{bal}_f(v)$ of vertex v with respect to f by

$$\mathrm{bal}_f(v) := \sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e).$$

The function f is called *conserving* at a vertex v if $\mathrm{bal}_f(v) = 0$.

The Maximum Flow problem asks to transport as many units from the source to the sink without violating the edge capacities. More precisely, a function $f : A \to \mathbb{R}^+$ is called an *s-t-flow* if:

(1) edge capacities are respected, i.e.,

$$0 \leq f(e) \leq c(e) \text{ for all } e \in A, \text{ and}$$

(2) f is conserving, i.e.,

$$\mathrm{bal}_f(v) = 0 \text{ for } v \in V - \{s, t\}, \quad \mathrm{bal}_f(s) \geq 0, \quad \text{and} \quad \mathrm{bal}_f(t) \leq 0.$$

Its *value* is defined by $\mathrm{val}(f) = \mathrm{bal}_f(s)$. See Figure 2.1.

Problem 2.1 Maximum Flow

Instance. A network $N = (G, c, s, t)$.

Task. Find an $s - t$-flow of maximum value in N.

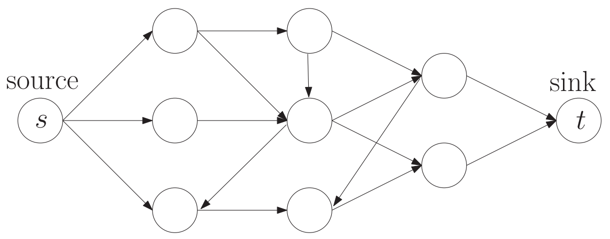


Figure 2.1: A network with source s and sink t.

We can formulate the maximum flow problem as an LP in the variables f_e for $e \in A$.

$$\begin{aligned} \text{maximize} \quad & \sum_{e\in\delta^+(s)} f_e - \sum_{e\in\delta^-(s)} f_e, \\ \text{subject to} \quad & \sum_{e\in\delta^+(v)} f_e - \sum_{e\in\delta^-(v)} f_e = 0 \quad v \in V - \{s,t\}, \\ & f_e \le c(e) \quad e \in A, \\ & f_e \ge 0. \end{aligned}$$

Since the flow $f = 0$ is feasible for this LP, and the LP is obviously bounded (by $\sum_{e\in\delta^+(s)} c(e)$) we have that the MAXIMUM FLOW problem always has an optimum solution. Of course, we can solve the problem by using any algorithm for solving LPs but we are not satisfied with this – we want a combinatorial algorithm (without solving an LP) with guaranteed polynomial running time.

Let S be a subset of the vertices, called a *cut*. The induced *cut-edges* is the set of *outgoing edges* $\delta^+(S) = \{uv \in A : u \in S, v \in V - S\}$ and *incoming edges* $\delta^-(S) = \{vu \in A : u \in S, v \in V - S\}$. Define its *capacity* by $\operatorname{cap}(S) = \sum_{e\in\delta^+(S)} c(e)$. An $s-t$-cut is a cut so that $s \in S$ and $t \in V - S$. A *minimum cut* refers to one with minimal capacity among all $s-t$-cuts. We extend the definition of *balance* also for any cut S:

$$\operatorname{bal}_f(S) = \sum_{e\in\delta^+(S)} f(e) - \sum_{e\in\delta^-(S)} f(e).$$

The following result tells us that the value of a flow can be expressed through the incoming and outcoming flow of an arbitrary cut. Furthermore, the value of any flow (including the maximum one) is bounded from above by the capacity of any cut. We will see soon that the value of a maximum flow equals the capacity of a minimum cut.

Lemma 2.1. *For any $s-t$-cut S and any $s-t$-flow f we have that*

(1) $\operatorname{val}(f) = \operatorname{bal}_f(S)$,

(2) $\operatorname{val}(f) \le \operatorname{cap}(S)$.

Proof. We use the flow conservation property, i.e., $\mathrm{bal}_f(v) = 0$ for all $v \in S - \{s\}$ to find

$$\begin{aligned}
\mathrm{val}(f) = \mathrm{bal}_f(s) &= \sum_{v \in S} \mathrm{bal}_f(v) \\
&= \sum_{v \in S} \left(\sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e) \right) \\
&= \sum_{e \in \delta^+(S)} f(e) - \sum_{e \in \delta^-(S)} f(e) \\
&= \mathrm{bal}_f(S).
\end{aligned}$$

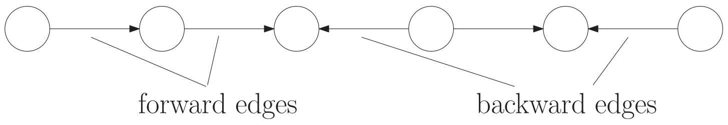
Furthermore we have $\mathrm{val}(f) \leq \sum_{e \in \delta^+(S)} c(e) = \mathrm{cap}(S)$ since $0 \leq f(e) \leq c(e)$. $\square$

The following definitions and structural result are the basis for an algorithm. A *path* $P = e_1, \ldots, e_\ell$ is a sequence of pairwise disjoint edges with common vertices, that is $e_i \in A$ such that $v_i v_{i+1} \in A$ or $v_{i+1} v_i \in A$ for $i = 1, \ldots \ell - 1$, $e_i \neq e_j$ for $1 \leq i < j \leq \ell$. The number ℓ of edges in P is called its *length.* A *v-w-path* P has the form $e_1 = v\cdot$ and $e_\ell = \cdot w$, i.e., it *starts* at v and *ends* at w. An edge $e = vw$ in a path is called *foreward edge* if $vw \in A$; *backward edge* if $wv \in A$. (A v-v-path is called a *cycle.*)

An s-v-path P is called *f-augmenting* with respect to a flow f if

(1) $f(e) < c(e)$ for every forward edge $e \in P$,

(2) $f(e) > 0$ for every backward edge $e \in P$.



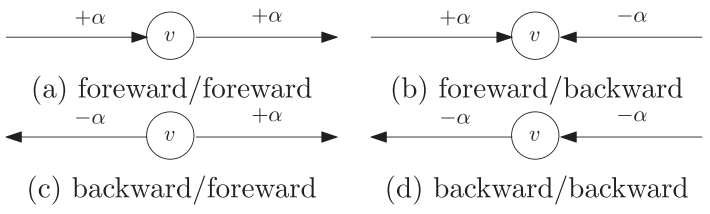
By how much can we increase the current flow value using a particular augmenting path P? Define the quantity

$$\alpha = \min\{c(e) - f(e) : e \text{ foreward edge in } P\} \cup \{f(e) : e \text{ backward edge in } P\}.$$

The following construction of a new flow f' is called *augmenting* f by α along P. Set $f'(e) = f(e) + \alpha$ if e is foreward edge in P, $f'(e) = f(e) - \alpha$ if e is backward edge in P, and $f'(e) = f(e)$ otherwise.

Observation 2.2. *The function f' defines a flow.*

Proof. By definition of the quantity α and because each edge occurs at most once in P, we have that $0 \leq f'(e) \leq c(e)$ for all $e \in A$. It remains to show that f' is flow conserving. It is clear that $\mathrm{bal}_{f'}(s) \geq \mathrm{bal}_f(s) \geq 0$ and consequently $\mathrm{bal}_{f'}(t) \leq \mathrm{bal}_f(t) \leq 0$. Consider an augmentation along edges $e_i e_{i+1}$ with $e_i = v_i v_{i+1}$ and $e_{i+1} = v_{i+1} v_{i+2}$ for $i = 1, \ldots, \ell - 1$. Call $v = v_{i+1}$ and distinguish four cases:



(a) foreward/foreward (b) foreward/backward
(c) backward/foreward (d) backward/backward

Algorithm 2.1 FORD-FULKERSON

```
Input.    Network $N = (G, c, s, t)$ with $c : A \to \mathbb{R}^+$.

Output.   $s - t$-flow $f$ of maximum value.

Step 1. Set $f(e) = 0$ for all $e \in A$.

Step 2. Find an $f$-augmenting path $P$. If none exists then return $f$.

Step 3. Compute

            $\alpha = \min\{c(e) - f(e) : e \text{ foreward edge in } P\} \cup \{f(e) : e \text{ backward edge in } P\}.$

        and augment $f$ by $\alpha$ along $P$. Go to Step 2.
```

This yields the claim. □

Theorem 2.3. *In a network N, the maximum value of an $s-t$-flow equals the minimum capacity of an $s-t$-cut.*

Proof. We show that an $s-t$-flow f has maximum value if and only if there is no f-augmenting path from s to t. In that case we will be able to find a minimum cut R with equal capacity.

Let there be an f-augementing path P from s to t, let α be as above and obtain f' by augmenting f by α along P. Observe that $\text{val}(f') > \text{val}(f)$, i.e., that f is not maximal.

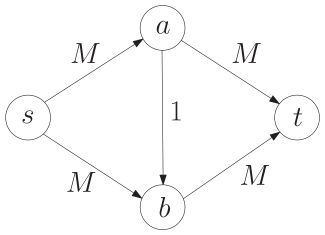
Now let there be no f-augmenting path from s to t. Consider the set S of vertices with augmenting paths from s, i.e., $S = \{v \in V : \text{there is an } f\text{-augmenting path from } s \text{ to } v\}$ and $t \notin S$. Thus S is an $s-t$-cut. By definition of augmenting paths, we must have $f(e) = c(e)$ for all $e \in \delta^+(S)$ and $f(e) = 0$ for all $e \in \delta^-(S)$. Hence, using Lemma 2.1 (1), we have $\text{val}(f) = \sum_{e\in\delta^+(S)} c(e) = \text{cap}(S)$. By Lemma 2.1 (2) f must be a maximum flow and S be a minimum cut. □

If all capacities are integers then α is an integer and the algorithm terminates after a finite number of iterations. Thus we obtain the following important consequence:

Corollary 2.4. *If the capacities of a network N are integers, then there is an integral maximum flow.*

If the capacities are not integers, then FORD-FULKERSON might not even terminate. Especially, we have not yet specified how we actually choose the augmenting paths mentioned in Step 2 of the algorithm. This must be done carefully in order to obtain a polynomial time algorithm as the following instance illustrates. It turns out that choosing shortest augmenting paths guarantee termination after a polynomial number of augmentations; see the EDMONDS-KARP algorithm.

Example 2.5. To show that FORD-FULKERSON is not a polynomial time algorithm consider the following network. Here M is a large number.



Alternatingly augmenting one unit of flow along the paths *s*-*a*-*b*-*t* and *s*-*b*-*a*-*t* requires $2M$ augmentations. This is already exponential because the (binary) input size of the graph is $O(\log M)$. In contrast the augmenting paths *s*-*a*-*t* and *s*-*b*-*t* already give a maximum flow after two augmentations.

It is an exercise to show the following *flow decomposition* result, which provides another structural insight into flows.

Theorem 2.6. *Given a network $N = (G, c, s, t)$ and an $s-t$-flow f then there is a familiy $\mathcal{P}$ of simple paths, a famlily $\mathcal{C}$ of simple cycles and positive numbers $h : \mathcal{P} \cup \mathcal{C} \to \mathbb{R}^+$ such that*

(1) $f(e) = \sum_{T \in \mathcal{P} \cup \mathcal{C} : e \in T} h(T)$ *for all* $e \in A$,

(2) $\operatorname{val}(f) = \sum_{T \in \mathcal{P}} h(T)$, *and*

(3) $|\mathcal{P}| + |\mathcal{C}| \leq |A|$.

2.2 Edmonds-Karp Algorithm

Example 2.5 suggests that it may be a good idea to always choose shortest augmenting paths, i.e., with minimum number edges. Indeed, the algorithm Edmonds-Karp below uses this strategy and yields polynomial running time.

Algorithm 2.2 Edmonds-Karp

Input. Network $N = (G, c, s, t)$ with $c : A \to \mathbb{R}^+$.

Output. $s-t$-flow f of maximum value.

Step 1. Set $f(e) = 0$ for all $e \in A$.

Step 2. Find a shortest f-augmenting path P w.r.t. the number of edges. If none exists then return f.

Step 3. Compute α as above and augment f by α along P. Go to Step 2.

Theorem 2.7. *The algorithm* Edmonds-Karp *computes a maximum $s-t$-flow f in any network N with n vertices and m edges in time $O\left(nm^2\right)$.*

The following lemma is crucial for the proof of the worst-case running time. Let $f_0, f_1, f_2, \ldots$ be the flows constructed by the algorithm. Denote the shortest length of an augmenting path from s to a vertex v with respect to f_k by $x_v(k)$ and respectively from v to t by $y_v(k)$.

Lemma 2.8. *We have that*

(1) $x_v(k+1) \geq x_v(k)$ *for all* k *and* v,

(2) $y_v(k+1) \geq y_v(k)$ *for all* k *and* v.

Proof. Suppose for the sake of contradiction that (1) is violated for some pair (v, k). We may assume that $x_v(k+1)$ is minimal among the $x_w(k+1)$ for which (1) does not hold.

Let e be the last edge in a shortest augmenting path from s to v with respect to f_{k+1}. Suppose $e = uv$ is a forward edge. Hence $f_{k+1}(e) < c(e)$, $x_v(k+1) = x_u(k+1) + 1$,and $x_u(k+1) \geq x_u(k)$ by our choice of $x_v(k+1)$. Thus $x_v(k+1) \geq x_u(k) + 1$. Suppose that $f_k(e) < c(e)$ which yields $x_v(k) \leq x_u(k) + 1$ and thus $x_v(k+1) \geq x_v(k)$, a contradiction.

Hence we must have $f_k(e) = c(e)$ which implies that e was a backward edge when f_k was changed to f_{k+1}. As we used an augmenting path of shortest length we have $x_u(k) = x_v(k) + 1$ and thus $x_v(k+1) - 1 = x_u(k+1) \geq x_u(k) \geq x_v(k) + 1$. Hence $x_v(k+1) \geq x_v(k) + 2$ yields a contradiction.

Similarly when e is a backward edge. The proof of (2) is analogous to (1). □

Proof of Theorem 2.7. When we increase the flow, the augmenting path always contains a *critical* edge, i.e., an edge where the flow is either increased to meet the capacity or reduced to zero.

Let $e = uv$ be critical in the augmenting path w.r.t. f_k. This path has $x_v(k) + y_v(k) = x_u(k) + y_u(k)$ edges. If e is used the next time in an augmenting path w.r.t. f_h, say, then it must be used in the opposite direction as w.r.t. f_k.

Suppose that $e = uv$ was a forward edge w.r.t. f_k. Then $x_v(k) = x_u(k) + 1$ and $x_u(h) = x_v(h)+1$. By Lemma 2.8 $x_v(h) \geq x_v(k)$ and $y_u(h) \geq y_u(k)$. Hence $x_u(h)+y_u(h) = x_v(h) + 1 + y_u(h) \geq x_v(k) + 1 + y_u(k) \geq x_u(k) + y_u(k) + 2$. Thus the augmenting path w.r.t. f_h is at least two edges longer than the augmenting path w.r.t. f_k. Similarly if e is a backward edge.

No shortest augmenting path can contain more than $n - 1$ edges and hence each edge can be critical at most $(n - 1)/2$ times. As each augmenting path contains at least one critical edge, there can be at most $O(nm)$ augmentations and each one takes time $O(m)$. This yields the running time of $O(nm^2)$. □

There are further algorithms that solve the Maximum Flow problem in less time. For example the Goldberg-Tarjan algorithm runs in time $O\left(n^2\sqrt{m}\right)$; with sophisticated implementations $O\left(nm\log(n^2/m)\right)$ and $O\left(\min\{m^{1/2}, n^{2/3}\}m\log(n^2/m)\log c_{\max}\right)$ can be reached.

2.3 Minimum Cost Flows

In this section we treat a more general problem than the Maximum Flow problem, namely the Minimum Cost Flow problem. We are again given a digraph $G = (V, A)$ with edge capacities $c : A \to \mathbb{R}^+$ and in addition to that a weight function $w : A \to \mathbb{R}^+$ indicating the *cost* of an edge. Thus a *network* is denoted $N = (G, c, w, b)$.

Now we define a modified notion of a flow. For any mapping $b : V \to \mathbb{R}$ with $\sum_{v\in V} b(v) = 0$ the value $b(v)$ is called the *balance* of a vertex v. If $b(v) > 0$ then v is called a *source*, if $b(v) < 0$ a *sink*. A *b-flow* in N is a function $f : A \to \mathbb{R}$ such that

(1) $0 \leq f(e) \leq c(e)$ for all $e \in A$ and

(2) $b(v) = \mathrm{bal}_f(v) = \sum_{e\in\delta^+(v)} f(e) - \sum_{e\in\delta^-(v)} f(e)$.

A 0-flow is called a *circulation.*

The *cost* of any flow f is

$$\mathrm{val}(f) = \sum_{e\in A} f(e)w(e).$$

Now the problem is to find a b-flow with minimum cost.

Problem 2.2 Minimum Cost Flow

Instance. A network $N = (G, c, w, b)$.

Task. Find an b-flow of minimum cost in N or decide that none exists.

The second part of our task is easy. Given a network $N = (G, c, w, b)$ with balance vector b, we can decide if a b-flow exists by solving a Maximum Flow problem: Add two vertices s and t and edges sv, vt with capacities $c(sv) = \max\{0, b(v)\}$ and $c(vt) = \max\{0, -b(v)\}$ for all $v \in V$ to N. Then any $s-t$-flow with value $\sum_{v\in V} c(sv)$ in the resulting network corresponds to a b-flow in the original network N.

For the remainder of the section we give an optimality criterion which leads directly to an algorithm similar to the Ford-Fulkerson method. But here we augment along cycles instead of paths. Again, the choice of the augmenting cycles must be done carefully. But we omit this here and state the following theorem which refers to Orlin's algorithm without proof.

Theorem 2.9. *There is an algorithm which solves the* Minimum Cost Flow *problem on any network with n vertices and m edges in time $O(m \log m(m + n\log n))$.*

We begin our discussion of an optimality criterion with a definition. Given a digraph $G = (V, A)$ with capacities c, weights w, and a flow f in G, construct the graph $R = (V, A + A_R)$ with $A_R = \{wv : vw \in A\}$, where $r \in A_R$ is called a *reverse* edge. (The notation "+" here means that we actually allow parallel edges in R). The *residual capacities* $c_R : A + A_R \to \mathbb{R}^+$ are $c_R(vw) = c(vw) - f(vw)$ for $vw \in A$ and $c_R(wv) = f(vw)$ for $wv \in A_R$. The *residual weight* $w_R : A \to \mathbb{R}$ is $w_R(vw) = w(vw)$ for $vw \in A$ and $w_R(wv) = -w(vw)$ for $wv \in A_R$. Finally define the *residual graph* $G_f = (V, A_f)$ with $A_f = \{e \in A + A_R : c_R(e) > 0\}$.

Now, given a digraph G with capacities c and a b-flow f, an *f-augmenting cycle* is a simple cycle in G_f. The following theorem is an optimality criterion for the Minimum Cost Flow problem.

Theorem 2.10. *Let $N = (G, c, w, b)$ be an instance of the* Minimum Cost Flow *problem. A b-flow f is of minimum cost if and only if there is no f-augmenting cycle with negative total weight.*

We prove the theorem in two steps. First we show that the difference between any two b-flows gives rise to a circulation and second that this circulation can be decomposed into circulations on simple cycles.

Lemma 2.11. *Let G be a digraph with capacities c and let f and f' be b-flows in (G, c). Construct R and G_f as above and define $g : A + A_R \to \mathbb{R}^+$ by $g(e) = \max\{0, f'(e) - f(e)\}$ for $e \in A$ and $g(e) = \max\{0, f(e') - f'(e')\}$ for all $e \in A_R$ with corresponding $e' \in A$. Then g is a circulation in R, $g(e) = 0$ for all $e \notin A_f$ and* $\mathrm{val}(g) = \mathrm{val}(f') - \mathrm{val}(f)$.

Proof. At each vertex $v \in R$ we have

$$\begin{aligned}\sum_{e\in\delta^+_R(v)} g(e) - \sum_{e\in\delta^-_R(v)} g(e) &= \sum_{e\in\delta^+_G(v)} (f'(e)-f(e)) - \sum_{e\in\delta^-_G(v)} (f'(e)-f(e)) \\ &= \sum_{e\in\delta^+_G(v)} f'(e) - \sum_{e\in\delta^-_G(v)} f'(e) - \left(\sum_{e\in\delta^+_G(v)} f(e) - \sum_{e\in\delta^-_G(v)} f(e)\right) \\ &= b(v) - b(v) = 0.\end{aligned}$$

so g is a circulation in R.

For $e \notin A_f$ consider two cases: If $e \in A$ then $f(e) = c(e)$ and hence $f'(e) \le f(e)$ which gives $g(e) = 0$. If $e = wv \in A_R$ then $e' = vw \in A$ and $f(e') = 0$ which yields $g(e) = 0$.

We verify the last statement

$$\mathrm{val}(g) = \sum_{e\in A+A_R} w(e)g(e) = \sum_{e\in A} w(e)f'(e) - \sum_{e\in A} w(e)f(e) = \mathrm{val}(f') - \mathrm{val}(f)$$

and the proof is complete. □

Lemma 2.12. *For any circulation f in a digraph $G = (V, A)$ there is a familiy $\mathcal{C}$ of at most $|A|$ simple cycles in G and for each $C \in \mathcal{C}$ a positive number $h(C)$ such that $f(e) = \sum_{C\in\mathcal{C}:e\in C} h(C)$.*

Proof. Follows from Theorem 2.6. □

Proof of Theorem 2.10. If there is an f-augmenting cycle C with weight $\gamma < 0$, we can augment f along C by some $\alpha > 0$ and get a b-flow f' with cost decreased by $-\gamma\alpha$. So f is not a minimum cost flow.

If f is not a minimum cost b-flow, there is another b-flow f' with smaller cost. Consider g as defined in Lemma 2.11 and observe that g is a circulation with $\mathrm{val}(g) < 0$. By Lemma 2.12, g can be decomposed into flows on simple cycles. Since $g(e) = 0$ for all $e \notin A_f$, all these cycles are f-augmenting and one of them must have negative total weight. □

2.4 Assignment Problem

A graph $G = (V, E)$ with vertex set $V = L \cup R$ ("left" and "right") is called *bipartite* if the edge set satisfies $E \subseteq \{\ell r : \ell \in L, r \in R\}$. An *assignment* (also called a *matching*) is a subset $M \subseteq E$ such that for every $v \in V$ in the graph $H = (V, M)$ we have $\deg_H(v) \le 1$. A matching is called *perfect* if $\deg_H(v) = 1$ for every $v \in V$. Of course, a necessary condition for the existence of a perfect matching in a bipartite graph is $|L| = |R|$.

The Assignment Problem has numerous applications and refers to the following. We are given a bipartite graph $G = (L \cup R, E)$ and a weight function $w : E \to \mathbb{R}$. We are asked to find a subset $M \subseteq E$ with minimum total weight, i.e.,

$$\mathrm{val}(M) = \sum_{e\in M} w(e),$$

such that M is a perfect matching or to conclude that no such matching exists.

Theorem 2.13. *The* Assignment *problem is a* Minimum Cost Flow *problem.*

Problem 2.3 Assignment

Instance. Bipartite graph $G = (L \cup R, E)$ and a weight function $w : E \to \mathbb{R}$.

Task. Find perfect matching M with minimum weight $\mathrm{val}(M) = \sum_{e \in M} w(e)$ or conclude that no such matching exists.

Proof. Let $G = (V, E)$ be a bipartite graph with $V = L \cup R$ and $|L| = |R| = n$. Now we construct a network N for the Minimum Cost Flow problem. We start with the vertices V, add a vertex s and connect it with every vertex $\ell \in L$ with directed edges $s\ell$. Further add a vertex t and introduce the directed edges rt for every $r \in R$. Further add directed versions of all edges $e \in E$, i.e., a directed edge ℓr is added for every undirected edge ℓr. The capacities of all these edges is one. The weights of the $s\ell$ edges and the rt edges are zero – the weights of the ℓr edges are equal to their weights in G. Now every integral b-flow f in N with $b = (b(s), b(v_1), \dots, b(v_n), b(t)) = (n, 0, \dots, 0, -n)$ corresponds to a perfect matching in G with the same weight, and vice versa. □

Below we give several applications of the Assignment problem. In most applications the requirement $|L| = |R|$ is disturbing, but can usually be handeled by adding artificial vertices and edges.

Bipartite Cardinality Matching

In the Bipartite Cardinality Matching problem we are given a bipartite graph $G = (V, E)$ with $V = L \cup R$, where $|L| \le |R|$. Our task is to find a matching with maximum number of edges. We construct a network similarly as before: we add vertices s and t and the directed edges $s\ell$ and rt for all $\ell \in L$ and $r \in R$. All these edges have capacity equal to one. Any integral $s - t$-flow of value k corresponds to a matching with k edges. Thus we have to solve a Maximum Flow problem.

Internet Dating

An internet dating website has ℓ females and r males in its pool. Furthermore, there is a preference system, where each person describes her/himself and her/his ideal partner. This system produces for each female i and each male j a value $w_{ij} > 0$. We seek to find an assignment of females to males with maximum total value. By adding dummy vertices with zero-weight edges to the appropriate side, and defining weights $-w_{ij}$ we arrive at an Assignment problem as defined above.

Scheduling on Parallel Machines

In the Scheduling on Parallel Machines problem we are given m machines and n jobs, where job j takes time p_{ij} if assigned to machine i. The jobs assigned to any machine are scheduled in a certain order. The completion time of job j is denoted c_j and refers to the following: If job j is assigned to machine i then the times p_{ik} of the jobs k also assigned to machine i but scheduled before job j contribute to the completion time of j, i.e., $c_j = \sum_{k \text{ on } i,\, k \le j} p_{ik}$ (where "$k \le j$" means that job k is scheduled before job j). The objective is to minimize the total completion time $\sum_j c_j$.

This problem can be formulated as an Assignment problem as follows: First consider the case that we have exactly one machine, i.e., all jobs have to be assigned to it. Consider a permutation π of $1, 2, \ldots, n$, where $\pi(j)$ gives the position of job j, and observe that we can write

$$\sum_{j=1}^{n} c_j = \sum_{j=1}^{n} (n - \pi(j) + 1) \cdot p_{1j},$$

because the contribution p_{1j} of job j in position $\pi(j)$ is counted $n - \pi(j) + 1$ many times in $\sum_j c_j$.

For multiple machines, the crucial observation is that the contribution of any job j can be described by p_{ij} times one of the multipliers $n, n-1, \ldots, 1$. Hence we create the following graph: A source s, a sink t, for each job a vertex v_j for $j = 1, \ldots, n$, and for each machine i exactly n slots, i.e., vertices s_{ik} for $k = 1, \ldots, n$. We add edges sv_j with zero weight and unit capacity. Furthermore, we add the edges $v_j s_{ik}$ with weight $(n - k + 1) \cdot p_{ij}$. Finally we add edges $s_{ik}t$ with zero weight and unit capacity. Any b-flow with $b = (b(s), b(v_1), \ldots, b(v_n), b(s_{11}), \ldots, b(s_{mn}), b(t)) = (n, 0, \ldots, 0, -n)$ with minimum cost corresponds to an optimal job-machine-assignment and vice versa.